1) Introduction

In this work, we explain how to train a multi-layered generative model of natural images. We use a dataset of millions of small color images, as described in the next section. Several groups have tried it without success. The models we focus on are RBM (Restricted Boltzmann Machine) and DBN (Deep Belief Network). These models teach interesting letters that we show are more useful for classics than raw pixels. We train Class Air on a composite subset of markers and call it the CIFAR-10 dataset.

Here we explain the algorithm for parallel training of RBMs. The algorithm is invaluable when both invisible and visible units are binary; it requires very little communication, so it measures very well. The algorithm is less old when the visible units are Gaussians, but it is almost simultaneous binary-to-binary in the problems we have. In both cases, the total amount of communication scales linearly with the number of machines, while the amount of communication required per machine is constant. If the machine has multiple cores, the work can be further subdivided. The specific algorithm we will describe is a distributed version of CD-1, but the principle is the same for any type of CD, including persistent CD (PCD).

2) The algorithm

Note that the (undivided) CD-1 training roughly consists of the following steps, where I

Visible units and J hidden units:

1. Get the data **V = [v0, v1, VI].**

2. Calculate the activity **H = [h0, h1, haj]** of the hidden data unit.

3. Visible unit activity (negative data) Count **V ′ = [v0 ′, v1, VI ′]** by looking at the hidden unit activity from the previous step.

4. Calculate the activity of the hidden unit with **H'= [h0', h1', hE]** according to the activity of the visible unit in the previous phase.

Feeling tired and constantly tired after losing weight.

We = ϵ Viejo - vi′hj.

When we train in batches, the weights are updated.

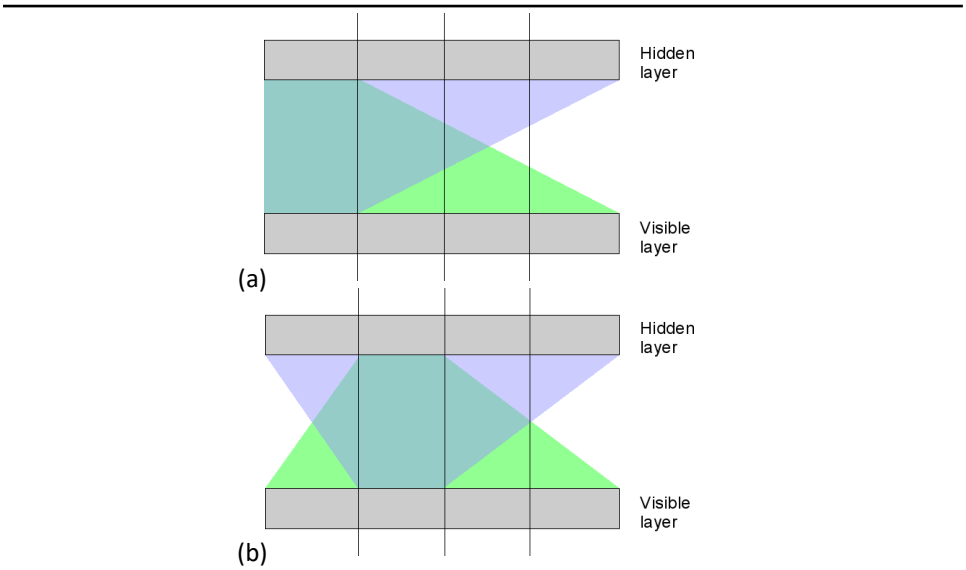
We = <Viejo> - <vi′hj ′>

Where <> represents expectations between training items.

The distributed algorithm performs steps 2, 3, and 4 in parallel and enters synchronization point 1 after each step so that all machines move forward in lockstep. If there are n machines, each machine is responsible for calculating 1/n unit activity hidden in phases 2 and 4 and 1/n unit activity shown in phase 3. We assume that all machines have access to the whole dataset. It's

probably easiest to describe this with a picture. Figure 4.1 shows which weights each machine cares about (i.e. which weights it has to know) when there are four machines in total.

For *K* machines, the algorithm proceeds as follows:



In a four-way parallel, it shows which scales (a) 1 and (b) 2 must be known. We randomly divided the visible and hidden layers into four equal parts. Note that the size of the visible layers does not have to be the same as the size of the hidden layers, so they are easy to draw. Weights come in different colors because they are easy to identify by their color in the text.

1. Each machine has access to the complete dataset and thus knows **V = [v0, v1, VI].**
2. Each machine k counts its hidden unit Hu = [he, HK, ..., he] to view the activity of data V. notes 0 1 J/k
3. That is, each machine only calculates hidden unit activity from J/K hidden units. It does this using.
4. Green weights in figure 4.1.
5. All machines send calculation results to each other in step 2.
6. All machines send calculation results to each other in step 4.
7. Now that all negative data is known, each machine calculates the activity of its hidden unit Hu ′ =

8. Use the green weights [he ′, HK ′, HK ′] from figure 4.1. 0 1 J/k
9. All machines send calculation results to each other in step 6.

Note that since there is only 1 bit per haj, the cost of providing hidden unit activity is very low. It is also cheaper to negotiate Weiss if the visible unit is also binary. Therefore, this algorithm works especially well for binary-to-binary RBMs.

After step 3, each machine will have enough data to calculate **<Viejo>** for all the weights it cares about. After step 7, each machine will have enough data to calculate **<vi′hj′>** for all the weights it cares about. That's all. You will notice that most weights are updated twice (but on different machines) because most of the weights are known on both machines. That is the value of parallelism. In fact, in our implementation, we update each weight twice, even if only one machine is aware of it. This is a relatively small amount of extra computation, and avoiding it would result in multiplying two smaller matrices instead of one larger one.

Beware of confusion at the starting point, and you might also ask: if every weight update is calculated twice, can we be sure that the same answer is calculated twice? When the visible unit is binary, we can. This is because the output point operation is formed by multiplying the data matrix by the hidden activity matrix, both of which are binary (although they are probably stored as output points). Although the visible units are Gaussian, this poses a problem. Since we have no control over the matrix multiplication algorithm that our numerical package uses to calculate matrix products, And since each weight can be stored in a different location of the weight matrix on each machine (and weight matrices can have slightly different shapes on different machines!), we cannot guarantee that the order of outgoing point operations will be the same as on one machine. we are the same, he calculates it on a different machine. Therefore, the stored values of these weights are separated on the two machines and we must introduce a weight synchronization phase in our algorithm. In practice, the bias is very small, even though all our calculations are done at one prep point, so the weight synchronization phase does not need to be performed often. The weight synchronization phase proceeds as follows: Set the purple weight (Figure 4.1) as the weight of the RBM, forgetting the green weight.

1. Each machine k sends part of its purple weight to machine k, which machine k needs to know to bring its green weight.

Note that each machine must send 1 part of the weight matrix to the K-1 machines. Since stu transmitting machines are K, the total amount of communication does not exceed K-1 (the size of the weight matrix). Therefore, the amount of communication required for weight synchronization is constant. Machines, meaning the output points don't break the algorithm, and as the number of machines increases, the algorithm doesn't need much precision.

Until now, we have avoided prejudice. The reason is that they are only a minor problem. Bias is divided into K-like weights. Each machine is responsible for updating the visible bias according to the visible units on the purple weights and the hidden bias associated with the units hidden on the green weights in Figure 4.1. Each anomaly is known only to one machine, so the updates only need to be calculated once. Note that CD-1 contains bias updates.

hydrogen bromide

= (<vi> - <vi ′>),

= ϵ <hj> - <h′j>

where eg stands for the bias of visible cell I and bh for the bias of invisible cell j. of course after the step. Any machine can be counted as <vi>. After step 2, the <hj> can count. It can be calculated after step 4.

<vi ′> and after step 6 it can count <h′j>

3) Implementation

We apply this algorithm to a Python program as a C extension. Python code starts matrices, communication channels, etc. The C code does the actual calculation. We use BLAS (Basic Linear Algebra Subroutines) to implement anything that can be used to perform matrix and vector operations. We use TCP sockets for communication between machines. Our implementation is also capable of dividing the work into multiple cores of the same machine in the same way as above. The only difference is that the communication between the cores is via shared memory instead of sockets. We use the threads' library to parallelize the kernels. We perform a post-workout weight sync phase on each batch of 8000 images. It turned out that the weights were adjusted to some precision within 10-9. Our implementation also calculates the speed and weight loss for all weights. When paralleling K machines, our implementation spawns K1 writer threads and K1 reader threads on every machine except the threads that do the computation (worker threads). The number of worker threads is a runtime variable, but it makes more sense to set it to the number of cores available on the machine.

4) Writer threads

Each Writer thread is responsible for sending data to another specific machine. There is one line for each author thread. Worker threads add items to these rows when the count is complete. The author arranges these items and sends them to his target machine. We separate communication from computation so that a machine that needs everything to continue computing doesn't have to slow down execution because it still has some data, you have to send it to another machine.

5) Reader threads

Each reader thread is responsible for receiving data from a different specific machine. Reader threads read data as it arrives from your target computer. This setting ensures that any attempt to send data is not blocked while the recipient is still using a computer. On the other hand, there is always a wire ready to receive. Likewise, since data is received as soon as it is available, no machine has to delay retrieving data just because it is still computing.

6) Results

We tested the implementation of our algorithm on a relatively large problem of 8,000 visible units and 20,000 hidden units. The training data consists of 8000 copies of random binary data (hopefully not to be confused with the fact that we have two different numbers of 8000). We ran this test on two dual-core Intel Xeon 3GHz CPUs. We use Intel's MKL (Math Kernel Library) for matrix operations. These machines have limited memory bandwidth, so when we start the fourth thread, the yields decrease. We measured our network speed at 105 MB/s, where

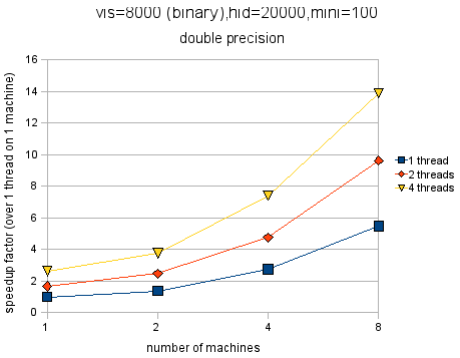
1 MB = 1,000,000 bytes.

We measured our network latency as 0.091 ms. Our network allows multiple machines to communicate at high speed without slowing down. In all our experiments, the time to complete the weight synchronization phase is no more than 4 seconds (with a single-shot accuracy of 2 seconds), and these times are included in all our angles. The RBM we studied here takes several minutes at best to train per batch, so the 4-second synchronization phase is incomprehensible. Acceleration factor for different numbers of machines and threads in parallel. 2 Note that these hoists do not show absolute speed, so it is not possible to compare the actual computation time of these RBMs on 8000 training samples between them (nor between sub-gears). Not surprisingly, double precision is slower than single precision, and smaller mini-batches are slower than larger ones. Single precision is very fast at first (about 3x), but as it gets more parallel it gets 2x faster.

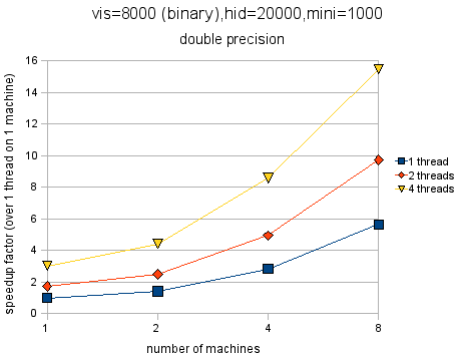
There are a few other things to consider.

- For binary-to-binary RBMs, in some cases it is better to have parallel courses between machines. Using a Gaussian to binary RBM is impossible, and the cost of inter-machine communication is reflected. Binary RBMs etc is also difficult when the mini-batch size is 100 because the accuracy of the algorithm depends on how often the machines have to talk to each other. Gores also points out that the algorithm scales better when using double-precision outliers rather than single-precision, although single-precision is still faster in all cases.
- For binary visible units, the algorithm scales linearly with the number of machines. Figure 4.6 illustrates this clearly. This shows that doubling the number of machines almost doubles the efficiency in all cases. The scattering algorithm scales slightly worse when the visible unit is Gaussian. Note that when the mini-batch size is 1000, 2 The number of individual threads running on a machine is calculated using different versions of Code One that do not tolerate all the overhead of parallelism. The number of multiple threads running on a machine is calculated using Code One version 3, which does not penalize the machine for parallelism (ie each weight update has to be calculated twice). When paralleled only with cores, not in a machine, there is an algorithm where each weight only needs to be updated once, and with only one core, while all other cores are updated immediately.

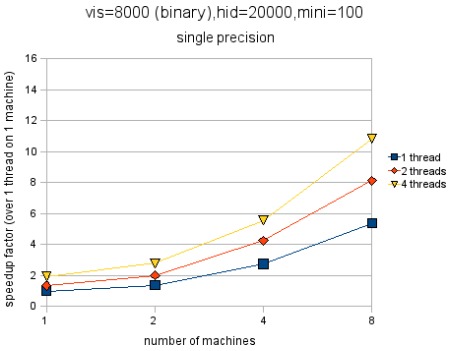
Come back after updating the new weights. This variable also reduces the number of sync points from three to two. The reason we don't show the acceleration factor using one machine versus two machines in these graphs is that the numbers for one machine are calculated using different codes (described in the previous footnote). Made and therefore not suitable To compare.



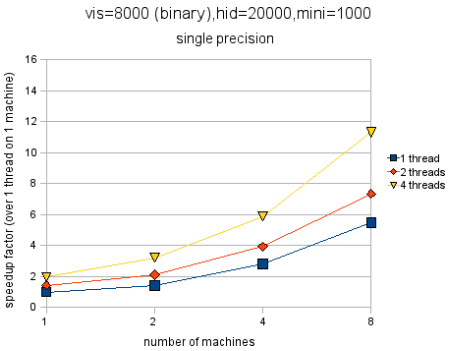
(a)



(b)



(c)



(d)

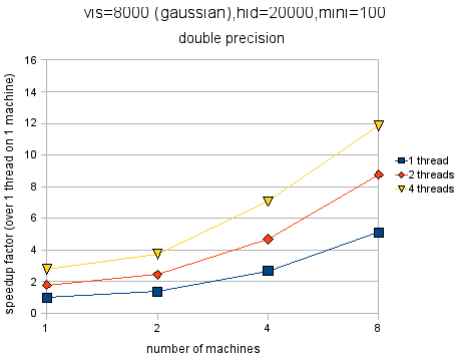
Speed through parallel-to-binary-to-binary RBMs (compared to threads running on the machine).

(a) Small batch of 100, double precision oats

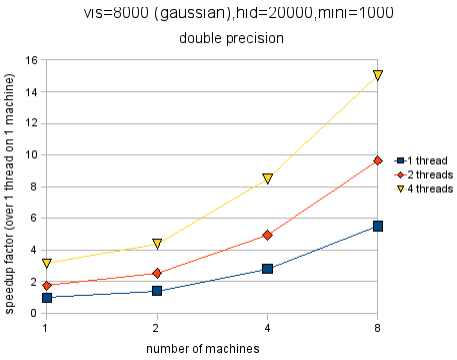
(b) Small batch of 1000, double precision Oatmeal

(c) Money Beach size 100, Single Precision oatmeal

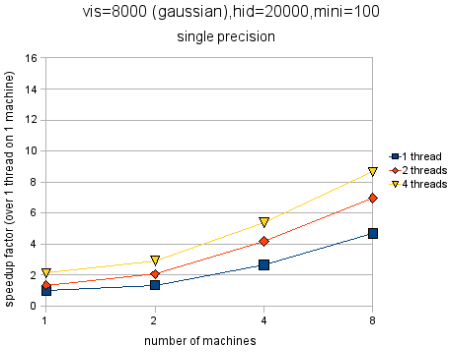
(d) Money Beach size 1000, Single Precision oatmeal



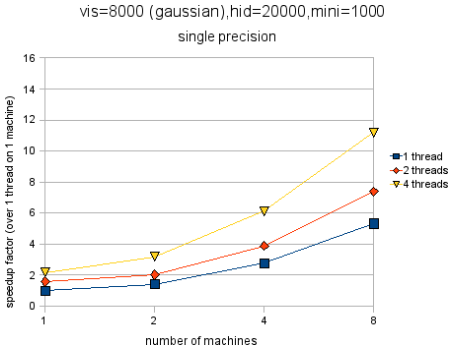
(a)



(b)



(c)



(d)

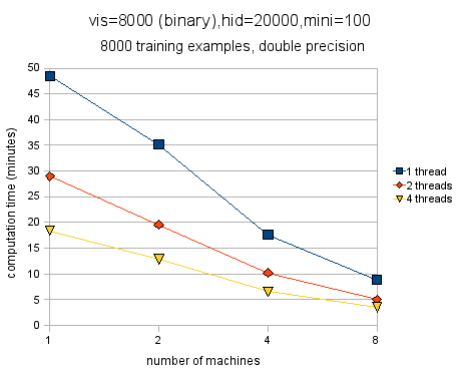
Figure: High speed through binary RBM to Gaussian parallelism (relative to threads running on the machine).

(a) Small batch of 100, double precision oats,

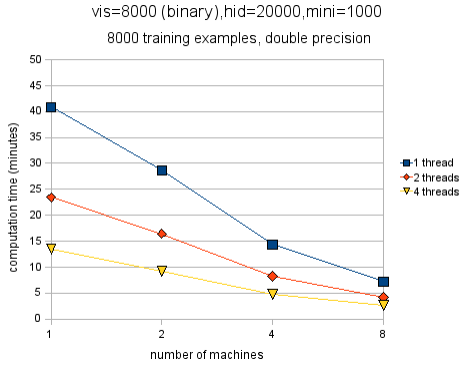
(b) Small batch of 1000, double precision Oatmeal,

(c) Money Beach size 100, Single Precision oatmeal,

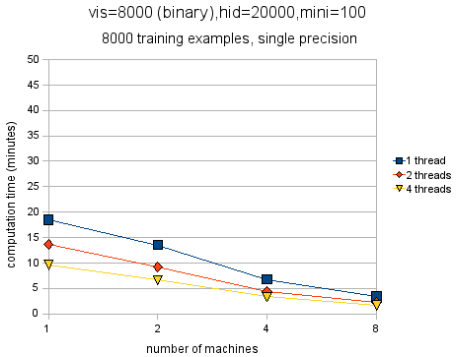
(d) Money Beach size 1000, Single Precision oatmeal



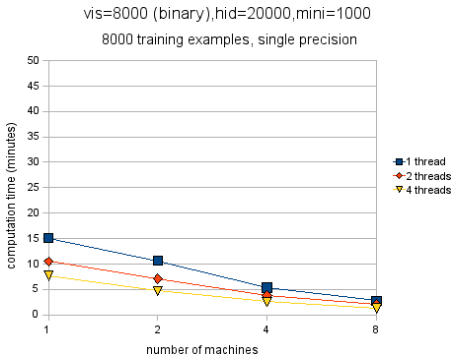
(a)



(b)



(c)



(d)

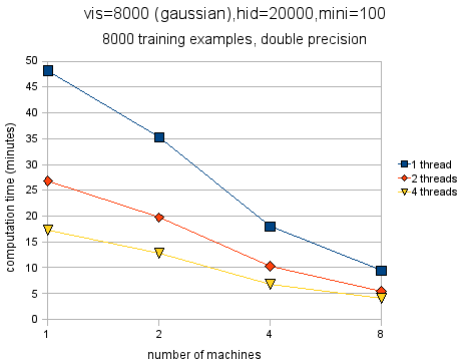
Figure: Training time on 8000 copies of random binary data (binary to binary RBM).

(a) mini-batch 100, double-precision oats,

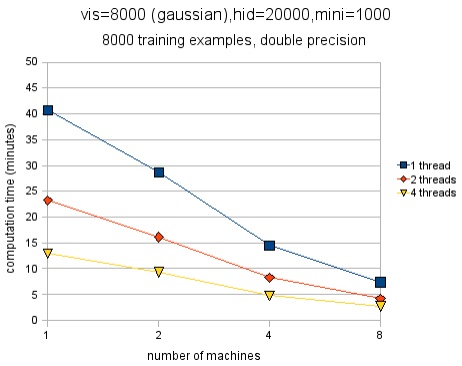
(b) mini-batch 1000, double-precision oats,

(c) mini-batch 100, single-precision oats

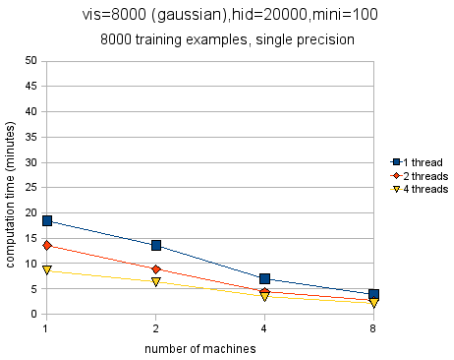
, (d) mini-batch 1000, single-precision oats



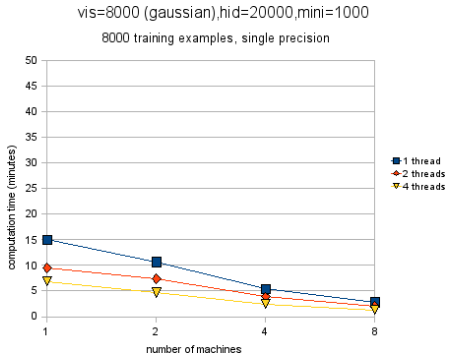
(a)



(b)



(c)



(d)

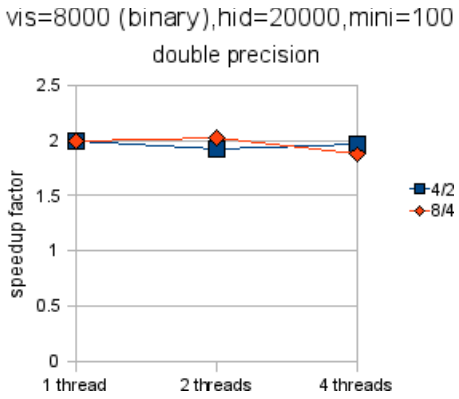
Training time on 8000 arbitrary real-valued data instances (Gaussian binary RBM).

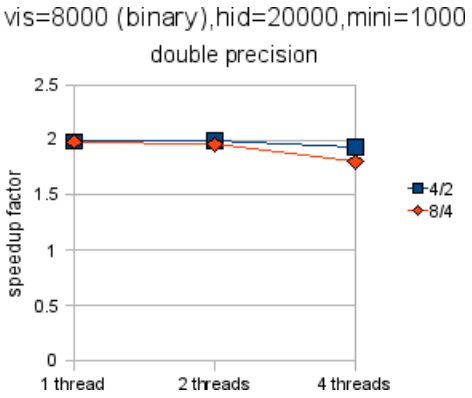
(a) mini-batch 100, double-precision oats,

(b) mini-batch 1000, double-precision oats,

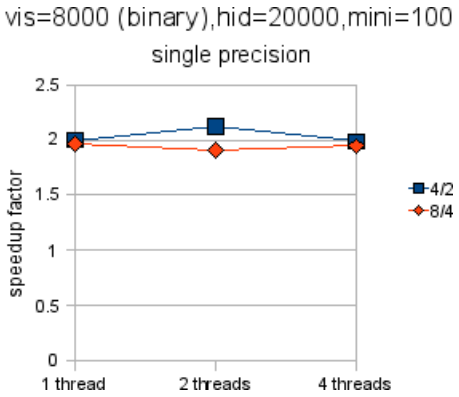
(c) mini-batch 100, single-precision oats,

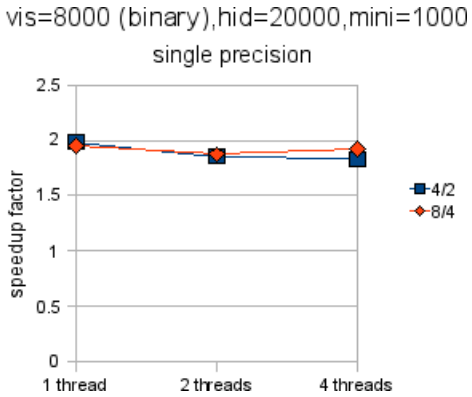
(d) mini-batch 1000, single-precision oats .





(a)





(b) **(d)**

The graph shows the acceleration factor when a binary number parallelizes a binary RBM in two (blue squares) with four machines and four (red diamonds) in eight machines. Note that doubling the number of machines almost doubles the efficiency. The algorithm scales from a

Gaussian in the binary case in almost the same way as in the binary-to-binary case. When the mini-batch size is 100, the binary version of the binary is better than that of Gaussian. Figure 4.7 shows that in the case of Gaussian to binary, the number of machines is doubled, but not enough, and the efficiency is doubled.

Communication cost analysis

We can analyze the communication costs more systematically. Looking at the K machines, we can calculate the amount of data transferred for each training batch of 8000 instances as follows: In steps 3 and 7, each machine must send its hidden unit activity to all other machines. This is a good thing and should end there.

2 • (K - 1) • 8000 20000 bits.

In step 5, each machine must send the activity of its visible cells to all other machines. If the visible unit is binary, it's a different one.

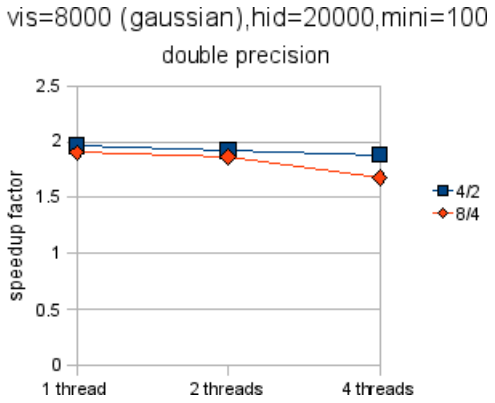
(K - 1) • 8000 8000 bits.

(K - 1) 3.84×108

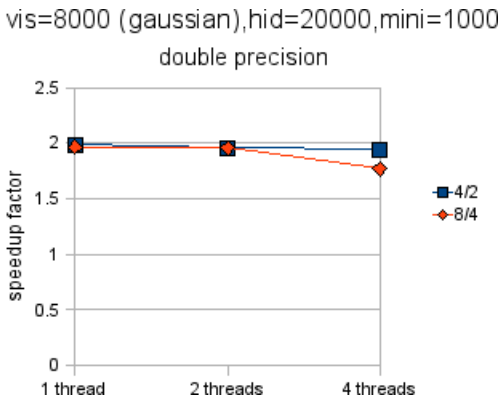
by

= 48 (K - 1) MB bits

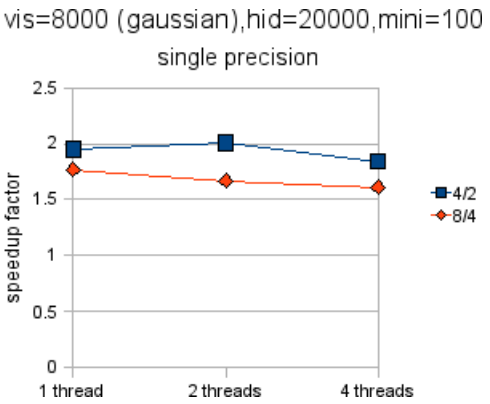
Thus the total amount of data sent (equivalently, received) by all machines is 48(K_1) MB. In the Gaussian-to-binary case, this number is appropriately larger depending on whether we're using single.



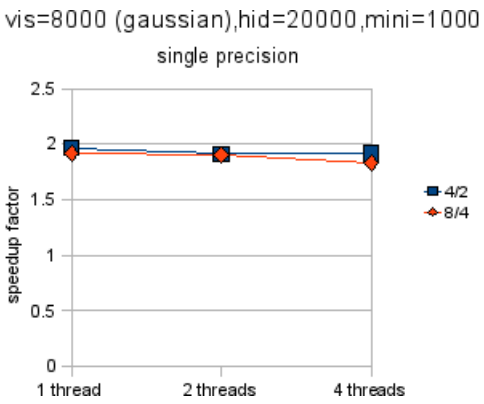
(a)



(b)



(c)



(d)

Figure: The graph shows the acceleration factor when aligning a Gauss to binary RBM on four machines versus two (blue squares) and eight machines versus four (red diamonds). Note that doubling the number of machines almost doubles the efficiency.

Several	1	2	4	8
Data sent (MB) (binary visible):	0	48	144	336
Data sent (MB) (Gaussian visible, single precision):	0	296	888	2072
Data sent (MB) (Gaussian visible, double precision):	0	552	1656	3864

The total amount of data sent through the RBM and discussed in the text (received evenly).

Or double-precision oats. In both cases, communication costs increase with the number of machines. Table 4.1 shows how much data is transferred through the trained RBM (excluding weight synchronization, which is very fast). Note that the communication cost per machine is related to a constant (48 MB in this case). Therefore, if machines can communicate with each other without reducing the communication of other machines, the communication cost does not increase with the number of machines. We don't seem to have reached the point where the communication costs outweigh the coordination benefits for this major problem. But when we used the algorithm to train an RBM on binary MNIST digits (784 visible units, 392 hidden units), after adding another machine, the advantage of synchronization disappeared. But at the time, each batch only lasted a few seconds.

7) Coding

1. Loading the dataset

```
import numpy as np

from keras.datasets import cifar10
from keras.utils.np_utils import to_categorical
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

2. Examining the dataset

```
print("Shape of training data:")
print(X_train.shape)
print(y_train.shape)
print("Shape of test data:")
print(X_test.shape)
print(y_test.shape)
import matplotlib.pyplot as plt
```

```
cifar_classes = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
print('Example training images and their labels: ' + str([x[0] for x in y_train[0:5]]))
print('Corresponding classes for the labels: ' + str([cifar_classes[x[0]] for x in y_train[0:5]]))

f, axarr = plt.subplots(1, 5)
f.set_size_inches(16, 6)

for i in range(5):
img = X_train[i]
axarr[i].imshow(img)
plt.show()
```

3. Preparing the Dataset

```
# Transform label indices to one-hot encoded vectors

y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)

# Transform images from (32,32,3) to 3072-dimensional vectors (32*32*3)

X_train = np.reshape(X_train,(50000,3072))

 X_test = np.reshape(X_test,(10000,3072))
 X_train = X_train.astype('float32')
 X_test = X_test.astype('float32')

 # Normalization of pixel values (to [0-1] range)

 X_train /= 255
 X_test /= 255
```

CNN Classifier

1. Preparing the dataset

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255

print("Shape of training data:")
print(X_train.shape)
print(y_train.shape)
print("Shape of test data:")
print(X_test.shape)
print(y_test.shape)
```

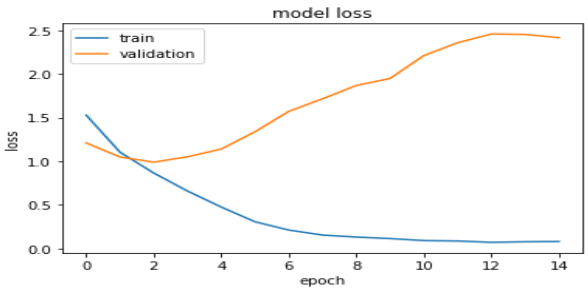
2. Creating CNN Model

```
from keras.layers import Dense, Flatten

from keras.layers import Conv2D, MaxPooling2D

model = Sequential()

model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))

model.add(Conv2D(32, (3, 3), activation='relu'))

model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
```

3. Training the CNN

```
history = model.fit(X_train, y_train, batch_size=32, epochs=15, verbose=2,
validation_split=0.2)
```

4. Plot Losses(history)



5. **Evaluating the CNN**

```
score = model.evaluate(X_test, y_test, batch_size=128, verbose=0)
print(model.metrics_names)
print(score)
```

8) Other algorithms

We will discuss some other algorithms that can be used for parallel RBM training. All of these algorithms require a lot of inter-machine communication, so they are not ideal for binary-to-binary RBMs. But for some problems, some of these algorithms require less communication than the algorithms we provide in the case of Gaussian to binary.

We are introducing an algorithm that performs a weight sync step after each weight update. Due to the frequent weight synchronization, the advantage is that each weight update does not have to be calculated twice, which saves me some calculations. Another advantage is that the activity of the hidden unit does not have to be discussed twice. However, in most cases, it will perform poorly due to the large amount of communication required in the weight synchronization stage. But if the mini-batch size is large enough, it will outperform the algorithm presented here. Assuming single-precision oats, this type requires less communication than the algorithm presented here in the following cases:

$mhK + 4mvK \geq 4vh$

where m is the size of the mini-batch, v is the number of visible units, h is the number of hidden units, and K is the number of machines. Note that the weights on the right are only twice the size of the matrix. So in general, when the weight matrix is smaller than the left side, the variables needless communicate. To see what this means, in the problem we are considering (v = 8000, h = 20000), the variant with K = 8 machines requires less communication than the algorithm we presented. When the size of the mini is large in batches. Note, however, that just adding machines up pushes the inequalities in several favorable directions, so it seems more attractive in terms of the number and complexity of machines another algorithm presented here is suitable for training RBMs with fewer hidden units than visible units. In this variant, each machine only knows 1/Kth of the data (data divided by dimensions, as before), where K is the number of machines. The algorithm is this:

1. With just a little bit of data, no machine can calculate the activity of the secret unit of a hidden unit. So each machine calculates the hidden unit input for all hidden units since it has 1/Kth of data. It has to do with purple weight.
2. Each machine sends 1/Kth of its input to another machine that the receiver cares about.
3. Once all machines have all the input they need to calculate the activities of their respective hidden units, they send all these activities to each other. These are binary values, so this step is cheap.
4. Count 1/Kth negative data per machine, given all hidden unit activity, again with purple weights.
5. Machine calculation of hidden unit activity due to negative data, as in steps 1-4.

The algorithm exchanges communication data to convey the hidden unit input. Roughly speaking, if the number of hidden units is less than half the number of visible units, then this is a win. Note that in this type each machine only needs to know the weight of the purple one. This means that each weight is only stored on one machine, so updates only need to be calculated once. The algorithm also avoids the weight bias problem discussed above due to a misunderstanding of the out point. A similar algorithm can be considered when there are more hidden than visible units. In this algorithm, each machine knows all the data, but only knows the green weight. In this way, no machine can recalculate the data individually, but they will work together by sending visible cell inputs to each other. Third, in a simple RBM training distribution algorithm, different machines are trained on different mini-batches and then send their weight updates to specific hosts. The machine then updates the average weights and sends the new weight matrix.

It doesn't take a lot of math to see that this algorithm will be very slow. The weight matrix must be sent 2 (K - 1) times per mini-batch. We tested this algorithm on the 8000 visible / 20000 hidden problem, and it turned out to be about 30% faster than the algorithm presented here, even though there are more hidden units than visible units in this problem. In our case, it seems that the cost of communication is not so high. This is the version of the algorithm we used to train the RBM.

9) Discussion & Conclusion

In this work, I propose three different convolutional neural network architectures for classifying images in the CIFAR10 dataset. The first two networks (layers 8 and 7, respectively) achieved commendable training and testing accuracies but dealt with severe overfitting in the third network, which utilized dropouts and alteration of L2 regularization. In addition, the third network achieves better authentication accuracy than the first two on the same number of rounds (20). The third network was trained on more visits, as the preliminary results from 20 visits promised higher accuracy for test validation.

Table 1. Summary: 3 Architecture Performance

- Architecture training
- Accuracy (%) Validation
- Accuracy (%) Test
- Accuracy (%) era #parameter
- Net worth I 86.18 72.39 71.81 20 193,034
- Net 2 95.37 67.00 67.07 20 686,090
- Net III 68.03 71.37 - 20 1,055,466
- Just three 73.21 76.60 75.43 40 1,055,466

10) References

Y. Lacuna, L. Bottom, Y. Bagnio, and P. Heffner, “Gradient-based learning applied to document recognition,” Proc. IEEE, vol. 86, no. 11, pp. 2278–2323, 1998.

A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks.”

“CIFAR-10 and CIFAR-100 datasets.” [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html. [Accessed: 20- Oct-2019].

D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” Dec. 2014.